

NEXT.JS

NEXT講座 基本編

この資料について



オンライン学習プラットフォーム『Udemy』
で公開している
『NEXT.js 基本講座』の説明用資料です

<https://www.udemy.com/course/nextjs-basic>

想定受講者



Next.js/TypeScriptをしっかり学びたい方
モダンなWeb開発方法を知りたい方
フロントエンド/バックエンドとともに
TypeScriptで書く方法を知りたい方
コード生成AIも活用してWeb開発をしたい方

講座作成の動機



生成AIの台頭によりコード生成の速度はかなり速くな
りましたが課題も見えてきました

- ・ コードのミス
- ・ 新旧混ざった情報(ライブラリ、記法の混在)
- ・ 意見がコロコロ変わる

人間による指摘や判断も不可欠だと実感

海外の動向



生成AIの発展によりFrontEndとBackEnd
をどちらも対応する場面が増えている

- ・ フルスタックエンジニア

どちらも同じ言語(React/TypeScriptなど)
で書く方が効率がいい



Next.js 概要

Next.js とは

Reactをベースにしたフレームワーク Web開発の効率を大幅に向上させるために設計されている Vercel社によってメンテナンス

公式HP <https://nextjs.org/>

特徴

クライアント側とサーバー側を1つのプロジェクトで管理できる

SSR/SSG/ISR/SPAを組み合わせていいとこどりができる

表示が早い・SEOにも強い(SSR)

事前にHTMLを生成できる(SSG)

自動的なルーティング コード分割(各ページごとに必要なコードだけ読み込む

先読み機能(Link Prefetching) 画像の最適化 静的キャッシュ

豊富なエコシステム 高速な環境構築 TypeScriptサポート

CSとSS 両対応



クライアントサイド
フロントエンド

~~NEXT~~.js

サーバーサイド
バックエンド



- ・ クライアントレンダリング
- ・ ルーティング
- ・ ステート管理
- ・ プリフェッчинг
- ・ 画像最適化

- ・ サーバーレンダリング
- ・ APIルート
- ・ データフェッчинг
- ・ サーバーアクション
- ・ ミドルウェア
- ・ 環境変数管理
- ・ DB連携

Next.js 簡易年表



2016年 ver1 リリース SSRの導入

2018年 ver7 webpackとbabelの自動設定機能

2019年 ver9 ファイルベースの動的ルーティング

2020年 ver10 画像最適化、国際化

2022年 ver13.0 app directoryとServerComponent

2023年 ver13.4 AppRouter正式版

2023年 ver14 ServerActions正式版, 部分プリレンダリング

2024年 ver15 Turbopack正式版

Googleトレンド その1



Googleトレンド その2



関連講座

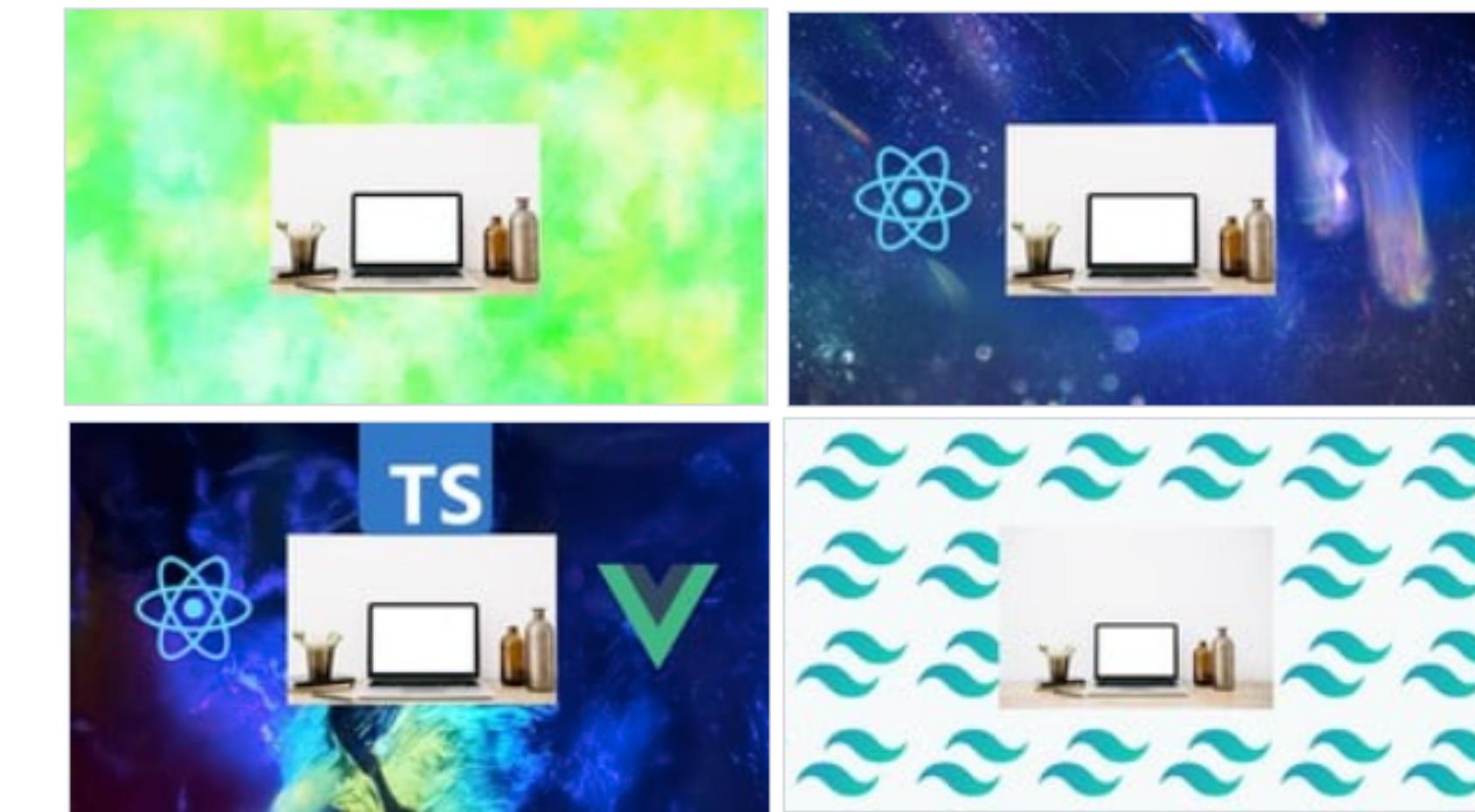
JavaScript (変数・関数・モジュール・イベント・async/awaitなど)

React (コンポーネント・hookなど)

TypeScript (基本的な型, ジェネリクス, typeなど)

TailwindCSS

クーポン一覧



https://note.com/aoki_monpro/n/nd695fc29792c



環境構築

環境構築



Node.js ver20以上推奨

デスクトップにnext-udemy-basicフォルダ

作成

ターミナルなどで

npx create-next-app@^15

インストール時の選択



What is your project name? .

Ok to processd? y

src/ directory? Yes

App Router? Yes

Turbopack? No

import alias? No

フォルダの内容

.next Next.jsのキャッシュや成果物を保存 自動生成

node_modules 依存パッケージを保存 npm installで自動生成

public 静的ファイル(画像、フォント、アイコン)などを保存 /から直接参照

src 開発フォルダ

.gitignore Gitで追跡しないファイルやフォルダを指定

eslint.config.mjs ESLintの設定ファイル

next-env.d.ts Next.jsがTypeScriptで動作するための型定義ファイル

next.config.ts Next.jsの設定ファイル

package-lock.json package.json 依存パッケージなどを管理する

postcss.config.mjs PostCSS(CSS処理ツール)の設定ファイル

README.md 説明書(Github トップページに表示)

tailwind.config.ts TailwindCSSの設定ファイル

tsconfig.json TypeScriptの設定ファイル

コマンド

npm run dev 開発モード ローカルサーバー

キャッシュは保存されない

<http://localhost:3000>

npm run build 本番向けビルド

.nextフォルダ生成(キャッシュ可能なリソースが生成)

npm run start 本番環境 稼働 (.nextが必要)(キャッシュ有効)

npm run lint コードの静的解析 build時にも実行

NEXT.JS

1. Get started by editing `src/app/page.tsx`.
2. Save and see your changes instantly.

▲ Deploy now

Read our docs

VS Code拡張機能



ES7 React/Redux/GraphQL/React-Native/JS snippets

Auto Import

Tailwind CSS IntelliSense

Prettify TypeScript (設定でmax depthを1->3に変更推奨)

Highlight Matching Tag

indent-Rainbow

アイコン、テーマ、Git関連はお好みで

(Dracula Theme, Git Graph, Material Icon Theme)



AppRouter ルーティング関連

appフォルダの内容



fonts フォント

favicon.ico フavicon

globals.css グローバルCSS

layout.tsx レイアウト指定ファイル

page.tsx トップページ

appフォルダ直下の整理



page.tsx

return内を<div>Home</div>だけにする

layout.tsx

fontを削除

globals.css

@tailwind以外の箇所を削除する

appフォルダ内の特別なファイル



フォルダ毎に設定可能 ファイル名は固定

page.tsx … ブラウザ表示ページ 自動ルーティング

layout.tsx … レイアウト

not-found.tsx … ページが見つからない

loading.tsx … ローディング

error.tsx … エラーページ

api/route.ts … APIエンドポイント(ルートハンドラー)

ルーティング



Next.js v13からはAppRouter推奨
(以前はPagesRouter)

src/app フォルダ内

フォルダ名・・URL

page.tsx ・・表示されるページの内容

ルーティング 基本



app/about/page.tsx

(VSCode拡張機能) rfc と入力してテンプレート作成
(importは不要なので削除)

localhost:3000/about で表示確認

```
export default function AboutPage(){  
  return (<div>AboutPage</div>  
)  
}
```

動的 ルーティング []



URLの一部が変動する localhost:3000/blog/10 などで表示確認

urlの情報取得・・・引数に {params} を指定 (オブジェクト形式)

propsやurlなどの情報を受け取る際は非同期関数を推奨

app/blog/[id]/page.tsx

```
export default async function BlogPage({params}){
```

```
    // console.log(params) ターミナルに内容表示
```

```
    const {id} = await params //分割代入
```

```
    return (<div>ブログID : {id}</div>
```

```
)
```

```
}
```

paramsの型指定

paramsはPromiseオブジェクト

◇・・・ジェネリクス 型を抽象化・柔軟にする

// Promiseのジェネリクスを省略するとanyとして扱われる

type Params = { //型を指定

 params : Promise<

 id: string

 }>

}

export default async function BlogPage({params}: Params) ...

ルーティンググループ()



URLには影響させたくないがまとめたい

app/(auth)/login/page.tsx

app/(auth)/register/page.tsx

URLとしては

localhost:3000/login,

localhost:3000/register

プライベートフォルダ



フォルダ名の先頭に _ をつけるとURLとして公開しない

フォルダ名 例

_components 共通コンポーネント

_utils ユーティリティ関数

_types 型定義

_api API関連

ex) _components/page.tsx

localhost:3000/_components で表示されない

app/layout.tsx

アプリ全体に影響するレイアウト htmlタグやbodyタグも記載

各ページの内容は { children } の箇所に表示される

用途: ヘッダー、ナビゲーション、フッター、メニュー

```
return (
  <html lang="ja">
    <body>
      <header className="bg-blue-200 p-4">Rootヘッダー</header>
      {children}
    </body>
  </html>
);
```

app/about/layout.tsx

下層ページでlayoutを作る際は RootLayout 引数内のコードをコピーして使う
{children}の箇所にpage.tsxのコンテンツが含まれる

```
export default function AboutLayout({ children }: Readonly<{ children: React.ReactNode; }>){  
  return (  
    <div className="bg-gray-100 h-screen">  
      {children}  
    </div>  
  )  
}
```

レイアウトは継承される



app/about/profile/page.tsx

aboutフォルダ直下のprofileにも背景色が継承される

```
export default function ProfilePage() {  
  return (  
    <div>  
      profile  
    </div>  
  )  
}
```

下層ページにも影響させたいかを考慮してフォルダ構成を決める必要がある

not-found.tsx カスタム404ページ



app/not-found.tsx

```
export default function NotFound(){  
  return (  
    <div>  
      そのページは存在しません  
    </div>  
  )  
}
```

loading.tsx



ページのデータ読み込み時に表示される スケルトンUIやローディングスピナーを定義
想定) ブロガー一覧ページを表示したら DBから各記事を取得して表示させる
取得するまでローディングと表示

app/blog/loading.tsx

```
export default function Loading = () => {  
  return (  
    <div>  
      Loding...  
    </div>  
  )  
}
```

app/blog/page.tsx その1

```
// ダミーデータ  
const articles = [  
  { id: "1", title: "タイトル1"},  
  { id: "2", title: "タイトル2"},  
  { id: "3", title: "タイトル3"},  
];
```

```
// 非同期関数内で待機 (awaitで待つにはPromise型を返す必要)  
async function fetchArticles(){  
  await new Promise((resolve) => setTimeout(resolve, 3000)); // 3秒待機  
  return articles;  
}
```

app/blog/page.tsx その2

```
// 配列を1件ずつ表示 (Reactではmapがよく使われる)
export default async function BlogPages(){
    const articles = await fetchArticles()

    return (<div>
        <ul>
            {articles.map(article => (
                <li key={article.id}>
                    title: {article.title}
                </li> )))
        </ul></div> )}
```

error.tsx



データ取得時にエラー発生した場合に表示 (クライアントコンポーネント)

app/blog/error.tsx

'use client'

```
export default function Error(){  
  return (  
    <div>  
      エラーが発生しました  
    </div>  
  )  
}
```

わざとエラーを出してみる



app/blog/page.tsx

略

```
async function fetchArticles(){  
    await new Promise((resolve) =>  
        setTimeout(resolve, 3000)); // 3秒待機  
        throw new Error('エラーが発生しました')  
        return articles;  
}
```

略

api/route.ts Route Handlers



APIルートを定義するファイル 別名 Route Handlers

サーバーサイドで動作する HTTPエンドポイントを作成できる

Next.jsではデータフェッチの方法が複数存在する

外部APIへの接続、複雑なデータ加工、認証、クライアントサイドからもアクセスがある場合に利用(CORS対応)

app/api フォルダを作成

その中に任意のフォルダ+route.tsを作成

app/api/hello/route.ts

```
// APIはdefault exportはできないので個別に設定  
// 関数名はGET, POST, PUT, PATCH, DELETE  
// NextResponseはResponseのラッパー  
// urlとしては http://localhost:3000/api/hello
```

```
import { NextResponse } from 'next/server'  
export async function GET(){  
    return NextResponse.json([  
        { id:1, name: '山田' },  
        { id:2, name: '田中' }  
    ])}
```

MetaData その1



メタデータ(SEO, SNSでのシェア表示(OGP), TwitterCards)

layout.tsxかpage.tsxに記載

metadata(静的) generateMetadata() (動的)

設定可能な項目

<https://nextjs.org/docs/app/api-reference/functions/generate-metadata#metadata-fields>

確認方法 ブラウザの開発ツールでheadタグを表示

MetaData その2

app/layout.tsx グローバル

```
import type { Metadata } from "next";
```

```
export const metadata: Metadata = {
  title: "Create Next App",
  description: "Generated by create next app",
};
```

app/blog/page.tsx ブログ一覧ページ (下層の情報があれば上書きされる)

```
import type { Metadata } from "next";
```

```
export const metadata: Metadata = {
  title: "ブログ一覧です",
  description: "ブログ一覧が表示されます",
};
```

MetaData その3



app/blog/[id]/page.tsx 個別ページ

```
export async function generateMetadata({params}: Params){  
  const {id} = await params  
  
  return {  
    title: `ブログ記事ID: ${id}です`  
  }  
}
```

ミドルウェア



ページ表示する前に実行したい処理を記述 ex)認証しているかどうか

src/middleware.ts

```
import { NextResponse } from 'next/server'
import type { NextRequest } from 'next/server'

export function middleware(request: NextRequest) {
  // ページリクエストのみに実行 ( image.jpgなどのドットは含まない)
  if (!request.nextUrl.pathname.includes('.')) {
    console.log('ミドルウェアのテスト')
  } return NextResponse.next() // リクエストを次の処理へ進める
}

export const config = {
  matcher: ["/blog/:path*"], // matcherで特定パスにのみ適用
};
```

このセクションのまとめ



Next.jsの環境構築

AppRouter環境での特別なファイル・フォルダ

特に重要 `page.tsx`, `layout.tsx`

フォルダ構成がそのままURLに適用される

下層にも影響する

フォルダ構成・階層を意識して開発を進める必要がある



コンポーネント (RCC/RSC)

クライアント側でJSを動かしすぎ問題



クライアントコンポーネント

Client

サイトをクリック



Server

HTML構築

← HTMLとJSを送信

データフェッチ

HTMLにデータを反映

サーバコンポーネント

Server

サイトをクリック



HTML構築

JSデータフェッチ

HTMLとJS

(データフェッチ以外の
処理)を送信



HTMLにデータを反映

サーバー側にもJSの処理を担ってもらい 描画速度UP



コンポーネント比較表 その1

	クライアントコンポーネント (RCC)	サーバーコンポーネント (RSC)
基本動作	'use client' ディレクティブをつける	AppRouter内のコンポーネントは 基本SC
バンドルサイズ	JSバンドルに含まれる	JSバンドルに含まれない
キャッシュ	クライアント側でのみキャッシュ	サーバー側でキャッシュ可能
useState/ useEffect	○	×
ブラウザAPI	○ (localStorage, window...)	×
イベントハンドラー	○ (onClick, onChange...)	×
DBアクセス	API経由	直接アクセス可能
ファイルシステム	×	○
環境変数	NEXT_PUBLIC_のみ	全てアクセス可能

コンポーネント比較表 その2

	クライアントコンポーネント (RCC)	サーバーコンポーネント (RSC)
初期ロード速度	バンドルサイズに依存	高速(JS不要)
サーバーリソース	クライアントリソースを使う	サーバー負荷あり
SEO	初期レンダリング時の考慮が必要	完全対応
使いわけ	フォーム入力 インタラクティブなUI アニメーション ブラウザAPIの使用 状態管理が必要な機能 クリックイベント処理	データ取得と表示 静的なUI SEO重視のコンテンツ 大きなライブラリの使用 DB操作 メタデータ設定

2種類のコンポーネント

app/rsc/page.tsx

```
export default function ServerComponent(){
  console.log('Server') // ターミナルに表示
  return (<div>Server</div>)}
```

app/rcc/page.tsx

'use client' // ディレクティブをつけるとRCCとして認識される

```
'use client'
export default function ClientComponent(){
  console.log('Client') // GoogleChrome開発ツールコンソールにも表示
  return (<div>Client</div>)}
```

RSC側でhookやeventはエラー

app/rsc/page.tsx

```
import {useState} from 'react'  
  
export default function ServerComponent(){  
  const [count, setCount] = useState(0);  
  
  console.log('Server')  
  
  return (  
    <div>  
      Server  
      <button onClick={() => setCount(count + 1)}>Count: {count}</button>  
    </div>  
  )  
}
```

This React hook only works in
a client component

RCC側ならOK

app/rcc/page.tsx

'use client'

```
import {useState} from 'react'
```

```
export default function ClientComponent(){
```

```
  const [count, setCount] = useState(0);
```

```
  console.log('Client')
```

```
  return (
```

```
    <div>
```

```
      Client
```

```
      <button onClick={() => setCount(count + 1)}>Count: {count}</button>
```

```
    </div>
```

```
  )
```

```
}
```

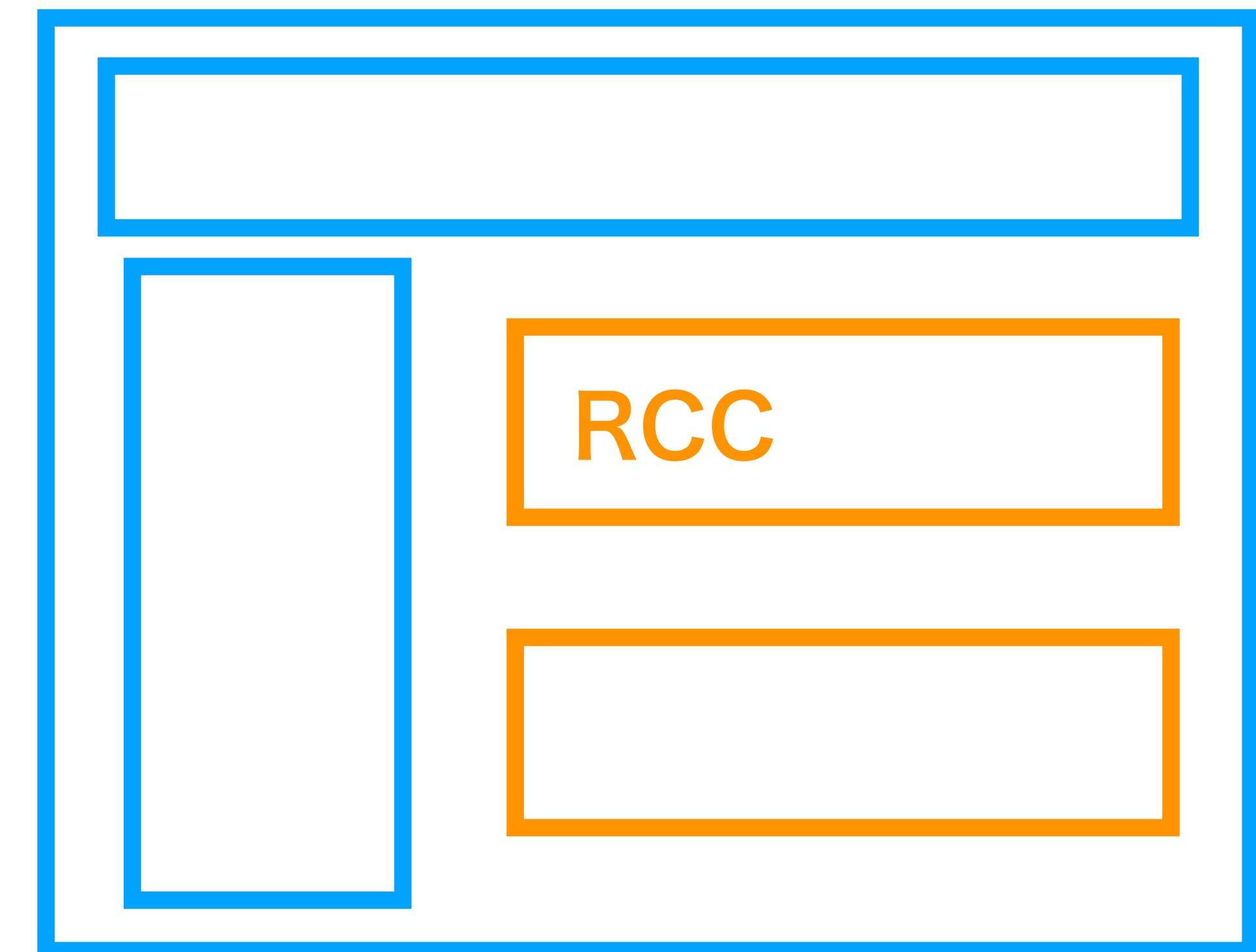
Rootヘッダー

ClientCount: 3

コンポーネントの組み合わせ

サーバーコンポーネントの中に
クライアントコンポーネントを
使う事もできる

RSC



Components



src/components/ClientComponent.tsxを作成

コード内容はクライアントのコードと同じ

app/rsc/page.tsx

```
import ClientComponent from '@/app/components/  
ClientComponent'
```

略

```
return (<div><ClientComponent /></div>)
```

RSCとRCCの使い分け

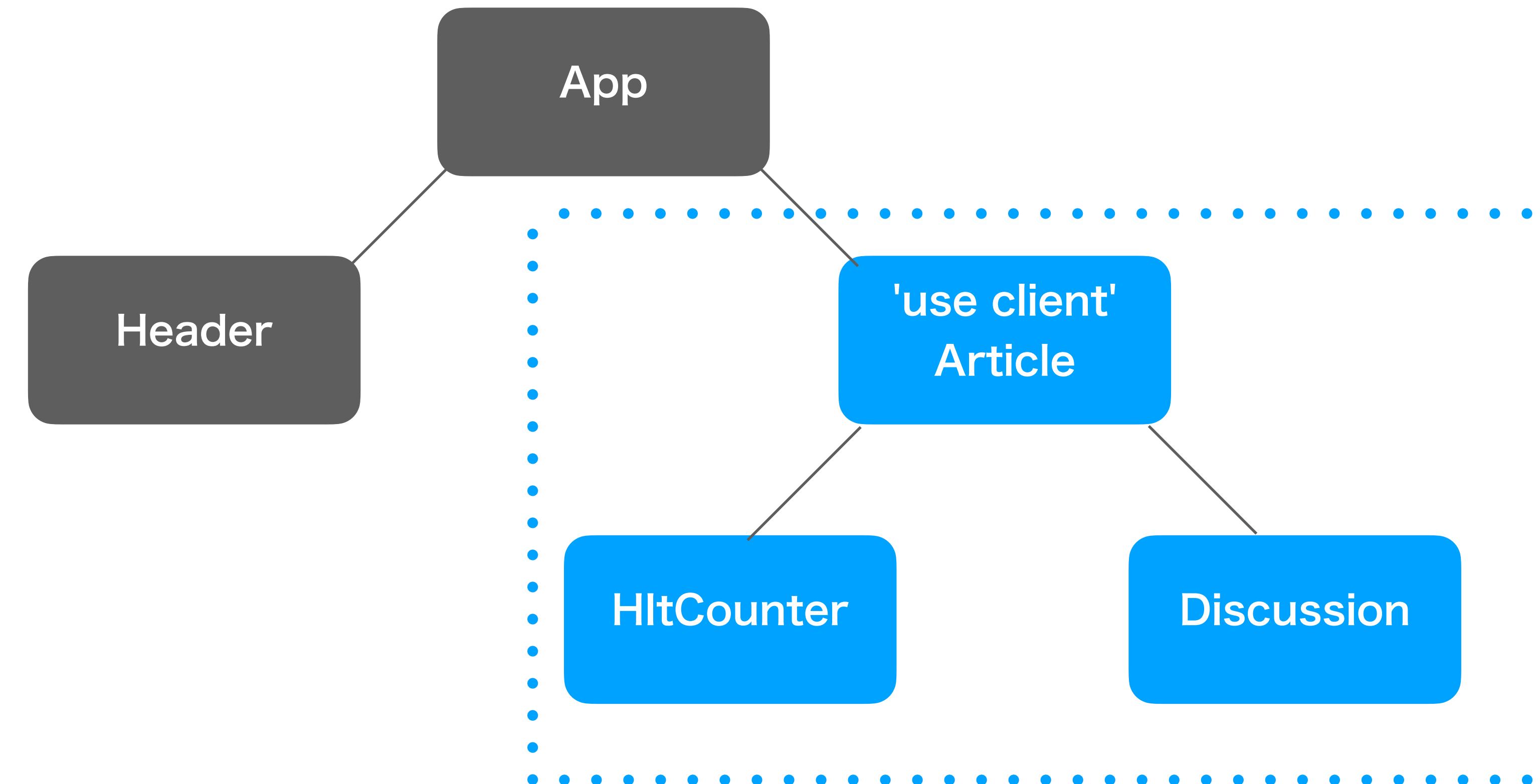


基本はRSC インタラクティブな機能の箇所だけRCC
大きなRSCの中に小さなRCCを配置する

データフェッチ
(画面描画時) RSCで実施 RCCにはpropsで渡す
(クリックなどユーザーアクション時) RCC

Client Boundary(境界)

use clientをつけるとその配下のコンポーネントは
自動的にRCCになる



リンクの使い分け

<a>タグは使わない・・・ページ全体がリロードされる ステートもリセットされる

RSC app/rsc/page.tsx 基本的なリンク

Linkコンポーネント (ページ遷移先を事前にプリフェッチし高速化)

```
import Link from 'next/link'
```

```
<Link href="/about">About</Link>
```

RCC app/rcc/page.tsx イベントも組み合わせるなら

```
import { useRouter } from 'next/navigation';
```

```
const router = useRouter()
```

```
<button onClick={()=> router.push('/about')}>About</button>
```

※ import {useRouter} from next/router は以前のPageRouter用なので使わない



レンダリング

レンダリング(画面描画)



Next.jsはコンポーネント毎にレンダリングを切り替え可能

CSR (Client Side Rendering)

クライアント側でレンダリング

SSR (Server Side Rendering)

サーバー側でレンダリング (常に更新)

SSG (Static Site Generation)

静的サイト生成 (ビルド時に生成 更新しない)

ISR (Incremental Static Regeneration)

特定の期間やタイミングで再更新

比較表

	CSR (クライアントサイド描画)	SSG (静的サイト生成)	ISR (増分 静的 更新)	SSR (サーバーサイド描画)
特徴	クライアントサイドでJSを使いHTMLを生成	ビルト時に静的なHTMLとして生成	SSGとCSRの中間 ビルト時に静的なHTML作成しキャッシュ 必要時のみ動的に更新	サーバー側でHTMLを動的に生成
メリット	動的にページを更新できる	SEOに強い 高速なページ表示	高速なページ表示 リアルタイム更新可能	SEOに強い 迅速な初期表示
デメリット	SEOに弱い 初期ロードが遅い	動的なコンテンツに不向き	サーバー負荷	サーバー負荷
fetchでデータ取得		fetch(URL, { cache: 'force-cache' });	fetch(URL, { next: { revalidate: 10 } }); // 10秒間隔でキャッシュ破棄し最新データ取得 export const revalidate = 10 (一定) revalidatePath() (任意のタイミング)	fetch(URL, { cache: 'no-store' });
データ取得がない fetch以外でデータ取得		デフォルト		export const dynamic = 'force-dynamic'
その他の指定方法		generateStaticParams() metadata, generateMetadata()	revalidateTag()	searchParams, headers(), cookies(), 動的なルートパラメータ
使い所	インタラクティブな箇所 リアルタイムな箇所	ビルト時にデータ取得	最新でなくてもいいが適度にデータを更新したい	常に最新のデータが必要なページ

画像最適化



Next.js のnext/image

画像の自動最適化(画面サイズに応じて最適なサイズにリサイズし、WebPなどの効率的なフォーマットに変換)

遅延読み込み(画面外の画像は必要な時のみに読み込まれパフォーマンス向上)

レスポンシブ対応(画像表示サイズを画面幅に合わせて自動調整)

Imageコンポーネントの設定

Next.js 画像最適化

```
import Image from 'next/image'
```

外部APIの画像使用時はnext.config.tsで外部ドメインを許可する必要

```
const nextConfig: NextConfig = {
  images: {
    remotePatterns: [
      {
        protocol: 'https',
        hostname: 'images.dog.ceo',
      },
    ],
  }
}
```

SSR



app/rendering/ssr/page.tsx

```
import Image from 'next/image'  
export const dynamic = 'force-dynamic' // SSR強制  
export default async function SSRPage() {  
  const res = await fetch('https://dog.ceo/api/breeds/image/random', {  
    cache: 'no-store'  
  })  
  const resJson = await res.json()  
  const image = resJson.message  
  const timestamp = new Date().toISOString()  
  return ( <div>SSR 每回リロード: {timestamp}  
    <Image src={image} width={400} alt="" />  
  </div>)  
}
```

SSG

app/rendering/ssg/page.tsx

```
// force-cacheでSSG指定もできる
```

```
export default async function SSGPage() {
```

```
  const res = await fetch('https://dog.ceo/api/breeds/image/random', {
```

```
    cache: 'force-cache'
```

```
)
```

```
  const resJson = await res.json()
```

```
  const image = resJson.message
```

```
  const timestamp = new Date().toISOString()
```

```
  return (<div>SSG ビルド時に生成され固定: {timestamp}
```

```
<Image src={image} width={400} alt="" />
```

```
</div>)
```

ISR

app/rendering/isr/page.tsx

```
import Image from 'next/image'  
export const revalidate = 10 // 10秒ごとに読み込み  
export default async function ISRPage() {  
  const res = await fetch('https://dog.ceo/api/breeds/image/random', {  
    next: { revalidate: 10 }  
  })  
  const resJson = await res.json()  
  const image = resJson.message  
  const timestamp = new Date().toISOString()  
  return ( <div>ISR 10秒ごとにリロード: {timestamp}  
    <Image src={image} width={400} alt="" />  
  </div>)  
}
```

コンパイル時に表示される

npm run build

```
↳ o /rendering/isr          <div>
↳ o /rendering/ssg          SSG ビルド
↳ f /rendering/ssr          </div>
```

○ (Static) 固定 (ISRは定期的に読み込み)

f (Dynamic) 都度レンダリング

npm run start

<http://localhost:3000/rendering/isr>

使い分け



コンポーネント分離することで組合せることもできる

SSG/ISR (静的・増分更新) ページのメインコンテンツ

SEO対策・高速な初期表示

SSR (毎回情報取得必要) ユーザー固有の部分

パーソナライズされたコンテンツ

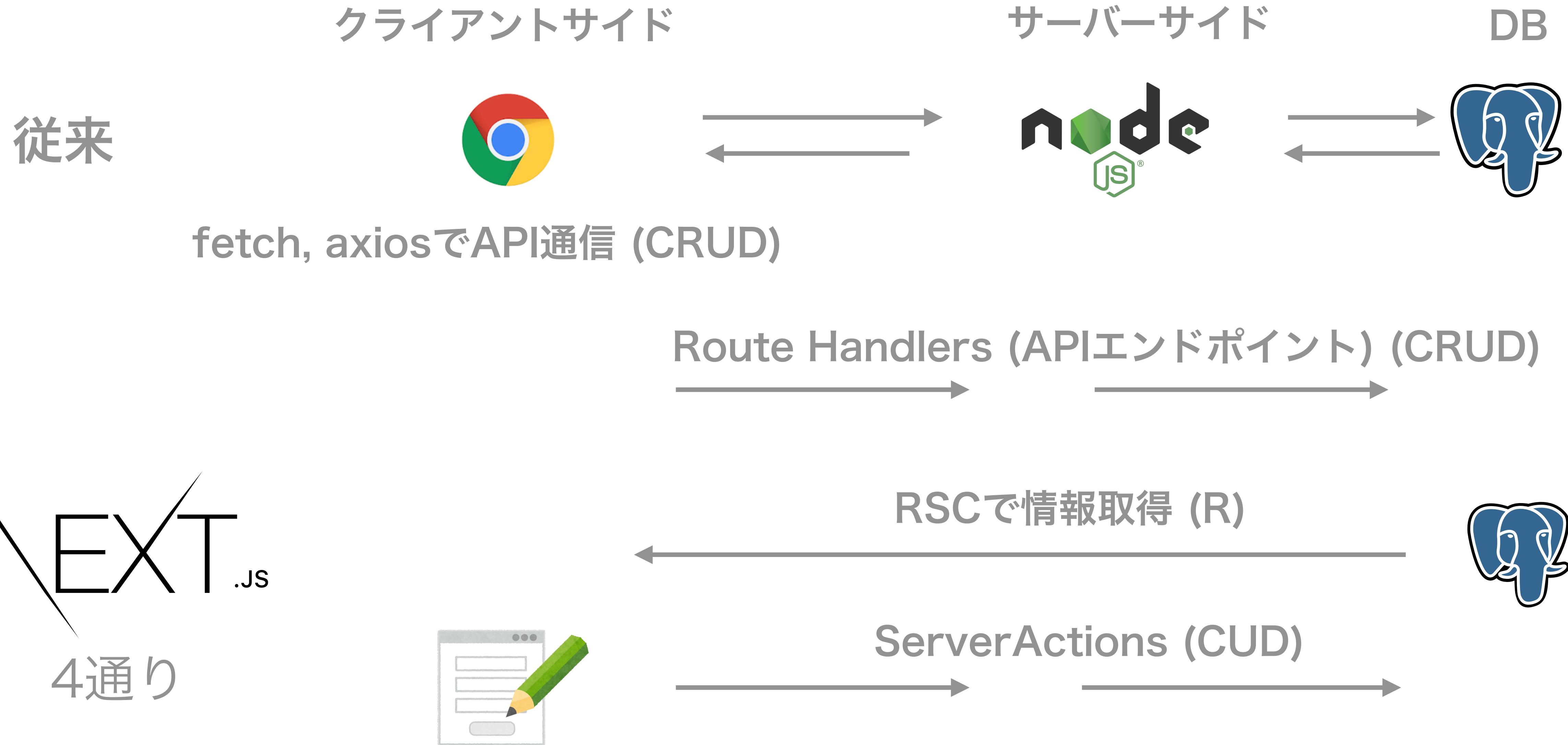
CSR インタラクティブな部分

フォーム、リアルタイム更新



データ操作方法の 比較

データ操作は4通り (CRUD)



4通りある理由



サーバー側にも処理を担ってもらう

パフォーマンスの最適化

SEO改善

セキュリティの向上

開発体験の向上

などの要因

比較表

	RCC (クライアントコンポーネント) API通信	APIルーティング機能 RouteHandlers	RSC (サーバーコンポーネント) 情報取得	ServerActions
実装場所	RCC 'use client'	app/api フォルダ	RSC	'use server'
主な用途	インタラクティブなUI、リアルタイム更新、動的データフェッチ	APIエンドポイント、外部API統合、複雑な処理、認証認可	データ取得、静的コンテンツ、SEO重視ページ、大規模データフェッチ	ユーザアクション(フォーム送信、データ更新、状態管理)
キャッシュ	SWR, TanStackQuery(useQuery)によるキャッシュ制御	手動設定必要	デフォルトで有効	キャッシュ制御可能 自動再検証
SEO	不向き	影響なし	最適 静的コンテンツ生成	最適 インタラクション維持
データ更新	リアルタイム可能、即時反映	エンドポイント依存 カスタム制御可能	再検証必要 Suspenseと統合可能	リアルタイム可能 最適化された更新
主要ライブラリ	useQuery, useSWR, axios, fetch API	Next.js 標準機能	fetch(Next.js拡張版), Prisma 等のORM, 直接DB接続可能	FormData, Zod, React Hook Form
メリット	リアルタイム更新、細かい制御可能、豊富なライブラリ、UI反応が早い	RESTful API作成、ミドルウェア使用可、柔軟な処理	ビルド時生成可能、型安全、セキュア、キャッシュ制御楽	バリデーション容易、フォーム処理に最適、JS不要
デメリット	初期表示遅延、JS必須、SEO不利、クライアントリソース消費	セットアップ工数、エンドポイント管理	動的更新は複雑、GET限定、クライアント状態は非反映	GET以外のリクエスト限定、エラーハンドリング複雑
ユースケース	ダッシュボード、チャット、リアルタイム機能、動的UI	認証処理、外部API統合、複雑なバックエンド	商品一覧、ブログ記事、静的ページ、大規模データ表示	フォーム送信、データ更新、ユーザー操作、状態変更

使い分け具体例 認証つきシンプルブログ

機能	実装例	
記事一覧/詳細	RSCからDB直接アクセス	
記事検索	ServerActions	フォームはServerActions
認証	Auth.js(NextAuth)	RouteHandler middleware.tsで保護
認証後の投稿/編集	ServerActions	フォームはServerActions

※もし チャット機能、プッシュ通知、いいねの即時反映、リッチテキストエディタ、AI支援機能
下書き自動保存、画像のアップロード進捗、ソート機能付き検索などの機能が必要な場合はRCC側の対応も必要

このセクションのまとめ



RCC/RSCの違いと使い分け

CSR/SSG/ISR/SSRの違い

データ操作方法は4通り

(RCC, RouteHandlers, RSC,

ServerActions)



ServerActions
を使った
フォーム送信

フォームに必要な要素



フォーム

バリデーション(クライアント、サーバー)

DB登録

扱うライブラリ

zod(型バリデーション) prisma(DB操作)

1. フォームページと完了ページ

```
// フォームページ app/contacts/page.tsx
import ContactForm from '@/components/ContactForm'
export default function ContactPage(){
  return (
    <div>
      <ContactForm />
    </div>
  )
}

// わかりやすくするために完了ページも作成 app/contacts/complete/page.tsx
export default function CompletePage(){
  return (
    <div>
      送信完了しました
    </div>
  )
}
```

1. フォーム作成 src/components/ContactForm.tsx (RCC)

```
'use client'

<form action="">

  <div className="py-24 text-gray-600">
    <div className="md:w-1/2 bg-white rounded-lg p-8 flex flex-col mx-auto shadow-md">
      <h2 className="text-lg mb-2">お問い合わせ</h2>
      <div className="mb-4">
        <label htmlFor="name" className="text-sm">名前</label>
        <input type="text" id="name" name="name" className="w-full bg-white rounded border border-gray-300 focus:border-indigo-500 focus:ring-2 focus:ring-indigo-200 outline-none py-1 px-3 leading-8" />
      </div>
      <div className="mb-4">
        <label htmlFor="email" className="text-sm">メールアドレス</label>
        <input type="email" id="email" name="email" className="w-full bg-white rounded border border-gray-300 focus:border-indigo-500 focus:ring-2 focus:ring-indigo-200 outline-none py-1 px-3 leading-8" />
      </div>
      <button className="text-white bg-indigo-500 py-2 px-6 hover:bg-indigo-600 rounded text-lg">送信</button>
    </div>
  </div>
</form>
```

2. ServerActions src/lib/actions/contact.ts

'use server'ディレクティブをつける事でServerActionsと認識される

```
'use server'

import { redirect } from 'next/navigation'

export async function submitContactForm(formData: FormData){
    const name = formData.get('name')
    const email = formData.get('email')

    // バリデーション
    // DB登録

    console.log('送信されたデータ:', {name, email})
    redirect('/contacts/complete')
}
```

2. ServerActionsをフォームに設定

src/Components/ContactForm.tsx

ServerActionsをインポートしてaction属性に設置

```
import { submitContactForm } from '@/lib/actions/contact'
```

```
export default function ContactForm(){
```

```
    return (
```

```
        <div>
```

```
            <h1>お問い合わせ</h1>
```

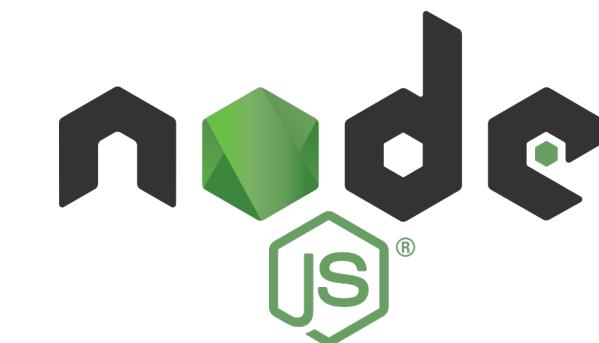
```
            <form action={submitContactForm}>
```

```
                <div>
```

バリデーション(検証)

入力情報に問題ないか検証

サーバーサイドは必須



クライアントサイドは100%検証できるとは限らない

(ブラウザにコードが表示され解除されるリスクが0ではない)

クライアントサイドもあるとUX向上

ex) サーバー送信前に入力不備、文字数オーバーなどがわかる

zod



TypeScript用のスキーマ検証(バリデーション)ライブラリ

<https://zod.dev/>

期待通りの形式になっているかをチェック

直感的な書き方

豊富なバリデーションルール

サーバー/クライアント両方で検証可能

インストール `npm install zod@^3`

バリデーション型

src/validations/contact.ts

```
import { z } from "zod"
```

```
export const ContactSchema = z.object({
  name: z.string()
    .min(3, "名前は3文字以上で入力してください")
    .max(20, "名前は20文字以内で入力してください"),
  email: z.string()
    .min(1, "メールアドレスは必須です")
    .email("正しいメールアドレスの形式で入力してください")
})
```

// 型の定義（フォームやAPIで使用）

```
export type ContactType = z.infer<typeof ContactSchema>
```

ServerActionsに追記

src/lib/actions/contact.ts

```
import { ContactSchema } from '@/validations/contact'
export async function submitContactForm(formData: FormData){
    // バリデーション
    const validationResult = ContactSchema.safeParse({ name, email })
    // バリデーションエラーの場合
    if (!validationResult.success) {
        // エラーメッセージの取得 flattenでエラーを見やすく加工
        const errors = validationResult.error.flatten()
        console.log('サーバー側でエラー', errors)
        return {}
    }
    // DB登録
```

RCC useActionState

React19で追加されたフック <https://ja.react.dev/reference/react/useActionState>

ServerActionの実行結果(成功/エラー)の状態管理

サーバーサイドでのバリデーションエラーの表示

サーバーサイドの処理中の状態管理

```
import { useState } from 'react'
```

```
const [state, formAction, isPending] = useActionState(サーバーアクション, {  
初期値})
```

state・・状態、formAction・・関数、isPending・・待機中

```
const [state, formAction] = useActionState(submitContactForm, {  
success: false, errors: {} })
```

<form action={formAction}> // 関数を差し替え

サーバーアクションの修正

```
// ActionStateの型定義
type ActionState = {
  success: boolean;
  errors: { name?: string[]; email?: string[]; };
  serverError?: string
}
export async function submitContactForm(prevState: ActionState, formData: FormData): Promise<ActionState>
```

```
// バリデーションエラーの場合
if (!validationResult.success) {
  // エラーメッセージの取得
  const errors = validationResult.error.flatten().fieldErrors //修正
  console.log('サーバー側でエラー', errors)
  return {
    success: false,
    errors: { name: errors.name || [], email: errors.email || [] } }
```

サーバー側のエラー表示



```
{ state.errors.name && (  
  <p className="text-  
  red-500 text-sm  
  mt-1">{state.errors.name.join(',')}</p>  
)}
```

クライアント側のエラー表示 1 (フォーカス外れた時)

```
import { useActionState, useState } from "react"
import { ContactSchema } from '@/validations/contact'
import { z } from 'zod'

const [clientErrors, setClientErrors] = useState({name: "", email: ""})

const handleBlur = (e: React.FocusEvent<HTMLInputElement>) => {
    const { name, value } = e.target

    // pickでスキーマを個別にピックアップし、parseでバリデーション
    try {
        if (name === 'name') {
            ContactSchema.pick({ name: true }).parse({ name: value })
        } else if (name === 'email') {
            ContactSchema.pick({ email: true }).parse({ email: value })
        }
    }
}
```

クライアント側のエラー表示 2 (フォーカス外れた時)

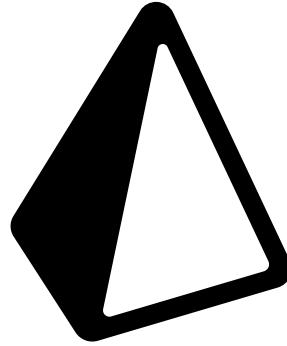
```
catch (error) {
    // バリデーションエラーかチェック
    if (error instanceof z.ZodError) {
        // 最初のエラーメッセージを取得
        const errorMessage = error.errors[0]?.message || ""
        setClientErrors(prev => ({
            ...prev, // スプレッド構文で既存のエラー状態をコピー
            [name]: errorMessage // 対象フィールドにエラーメッセージを設定
        }))
    }
}

<input type="text" id="name" name="name" onBlur={handleBlur} />
{clientErrors.name && (
    <p className="text-red-500 text-sm mt-1">{clientErrors.name}</p>)}
```



Prisma

Prisma



JS/TSのORM (Object-Relational Mapping)

<https://www.prisma.io/>

プログラミング言語でDBを操作 様々なDBを同じように記述できる
(sqlite, PostgreSQL, MySQL...)

特徴 型安全性, スキーマ定義でコード自動生成, 直感的
DBテーブルの作成、DB操作、ダミーデータの投入など
マイグレーション・・ DBテーブル編集の履歴
シード・・ 初期値・ダミーデータ

Prisma インストール



```
npm install prisma@^6
```

```
npm install @prisma/client@^6
```

Prismaの設定



npx prisma init // Prismaの初期化

prisma/schema.prismaと.envが生成される

prisma/schema.prismaの設定

provider= "sqlite" に変更 (ファイルベース 開発向けDB)

.env ファイル

postgresqlはコメントアウト

DATABASE_URL="file:./dev.db" を追記

prisma/schema.prisma (DBの構造)

データベース内 テーブルの列や属性を定義

@id • • PK @default • • デフォルト値 @unique • • 重複不可

? • • 必須ではない @updatedAt • • 更新年月

```
model Contact {  
    id      String  @id @default(cuid())  
    name    String  
    email   String  @unique  
    createdAt DateTime @default(now())  
    updatedAt DateTime @updatedAt  
}
```

マイグレーション



// マイグレーション(テーブル作成)

npx prisma migrate dev --name init

// DBの内容を確認

npx prisma studio

// DBリセット

npx prisma migrate reset

Prisma Studio

localhost:5555 でブラウザ確認

レコードのCRUDなど可能

The screenshot shows a web browser window with the URL 'localhost:5555'. The main title bar says 'Contact'. Below it, there's a toolbar with buttons for 'C' (refresh), 'Filters' (set to 'None'), 'Fields' (set to 'All'), 'Showing 0 of 0', and a prominent 'Add record' button. The main area displays five columns: 'id' (with a sorting arrow), 'name' (with a sorting arrow), 'email' (with a sorting arrow), 'createdAt' (with a sorting arrow), and 'updatedAt' (with a sorting arrow). Each column has a small checkbox icon at its top left.

Prisma クライアントインスタンス

lib/prisma.ts

```
import { PrismaClient } from '@prisma/client'
```

```
// グローバルスコープでPrismaインスタンスを保持できる場所を作る
```

```
const globalForPrisma = globalThis as unknown as {  
    prisma: PrismaClient | undefined  
}
```

```
// Prismaインスタンスがあれば使う、なければ作成
```

```
export const prisma = globalForPrisma.prisma ?? new PrismaClient()
```

```
// 開発環境でのみ使用
```

```
if (process.env.NODE_ENV !== 'production') globalForPrisma.prisma = prisma
```

参考: <https://www.prisma.io/docs/orm/prisma-client/setup-and-configuration/databases-connections#prevent-hot-reloading-from-creating-new-instances-of-prismaclient>

DBに登録

prisma.モデル名.メソッド でDB操作

lib/actions/contact.ts

```
import { prisma } from '@/lib/prisma'

const name = formData.get('name') as string
const email = formData.get('email') as string
// すでにメールアドレスが登録されているか確認
const existingRecord = await prisma.contact.findUnique({
  where: { email: email } })
if (existingRecord) {
  return { success: false,
    errors: { name: [], email: ['このメールアドレスはすでに登録されています'] }
  }
}

// 新規登録
await prisma.contact.create({
  data: { name, email }
})
```

DBに登録（詳細版）DBエラー時に返却

```
import { prisma } from '@/lib/prisma'
import { Prisma } from '@prisma/client'

const result = await prisma.contact.create({ data: { name, email } })
    .catch(error => {
        if (error instanceof Prisma.PrismaClientKnownRequestError) {
            if (error.code === 'P2002') { // ユニーク制約違反
                return {
                    success: false, errors: {},
                    serverError: 'このメールアドレスは既に登録されています' } as ActionState
            } } else {
                // その他のエラー
                return { success: false, errors: {}, serverError: 'サーバーエラーが発生しました' }
            } as ActionState
        }})
        // エラーチェックを追加
        if (result && 'serverError' in result) { return result}
```

RCC側にも表示させるなら



```
{ state.serverError && (  
  <p className="text-  
    red-500 text-sm  
    mt-1">{state.serverError}</p>  
)}
```

DB情報の表示

lib/contact.ts (モデルと対応させると整理しやすい)

参考: <https://www.prisma.io/docs/orm/prisma-client/queries>

```
import { prisma } from '@/lib/prisma';
```

```
export async function getContacts() {
  return await prisma.contact.findMany({
    select: { id: true, name: true, email: true, },
    orderBy: { createdAt: 'desc' },
  })
}
```

```
export async function getContact(id: string) {
  return await prisma.contact.findFirst({
    select: { name: true, email: true } })
}
```

画面表示 app/contacts/list/page.tsx

```
import { getContacts, getContact} from '@/lib/contact'
```

```
export default async function ListPage() {
    const contacts = await getContacts()
    const first = await getContact('1')
    return (
        <div>
            複数
            <ul>
                {contacts.map((contact)=>(<li key={contact.id}>{contact.name}: {contact.email}</li>))
            )
        </ul>
        1件
        <div>{ first ? first.name : '登録されていません'}</div>
        </div>
    )
}
```

このセクションのまとめ



ServerActionsを使ったフォーム送信

zod (型バリデーション)

サーバーサイドバリデーション (useActionState)

クライアントサイドバリデーション (useState)

Prisma (JS/TSのORM) DB操作